Tutorial do PostgreSQL 8

°

Projeto de Tradução para o Português do Brasil

(http://sourceforge.net/projects/pgdocptbr/)

°

The PostgreSQL Global Development Group

°Rio de Janeiro, 29 de dezembro de 2005

Documentação do PostgreSQL 8.0.0

Projeto de Tradução para o Português do Brasil (http://sourceforge.net/projects/pgdocptbr/)
The PostgreSQL Global Development Group
Copyright°©°1996-2005 por The PostgreSQL Global Development Group

Legal Notice

Traduzido por

- Halley Pacheco de Oliveira (<halleypo@users.sourceforge.net>)

Revisado por

- Diogo de Oliveira Biazus
- Marcia Tonon

Tutorial do PostgreSQL 8.x

Bem vindo ao Tutorial do PostgreSQL. Os poucos capítulos a seguir têm por objetivo fornecer uma introdução simples ao PostgreSQL, aos conceitos de banco de dados relacional e à linguagem SQL, para os iniciantes em qualquer um destes tópicos. Somente é pressuposto um conhecimento geral sobre a utilização de computadores. Nenhuma experiência com Unix ou em programação é necessária. Esta parte tem como objetivo principal fornecer experiência prática sobre os aspectos importantes do PostgreSQL. Não há nenhuma intenção em dar-se um tratamento completo ou abrangente dos tópicos cobertos.

Após ler este tutorial pode-se prosseguir através da leitura da Parte II para obter um maior conhecimento formal da linguagem SQL, ou da Parte IV para obter informações sobre o desenvolvimento de aplicativos para o PostgreSQL. Aqueles que instalam e gerenciam seus próprios servidores também devem ler a Parte III.

Capítulo 1. Início

1.1. Instalação

Para que se possa usar o PostgreSQL é necessário instalá-lo, obviamente. É possível que o PostgreSQL já esteja instalado na máquina, seja porque está incluído na distribuição do sistema operacional [1], ou porque o administrador do sistema fez a instalação. Se este for o caso, devem ser obtidas informações na documentação do sistema operacional, ou com o administrador do sistema, sobre como acessar o PostgreSQL.

Não havendo certeza se o PostgreSQL está disponível, ou se pode ser utilizado para seus experimentos, então você mesmo poderá fazer a instalação. Proceder desta maneira não é difícil, podendo ser um bom exercício. O PostgreSQL pode ser instalado por qualquer usuário sem privilégios, porque não é necessário nenhum acesso de superusuário (root).

Se for instalar o PostgreSQL por si próprio, então leia o Capítulo 14 para conhecer as instruções de instalação, e depois retorne para este guia quando a instalação estiver terminada. Certifique-se de seguir de perto a seção sobre a configuração das variáveis de ambiente apropriadas.

Se o administrador do sistema não fez a configuração da maneira padrão, talvez seja necessário algum trabalho adicional. Por exemplo, se a máquina servidora de banco de dados for uma máquina remota, será necessário definir a variável de ambiente `PGHOST` com o nome da máquina servidora de banco de dados. Também, talvez tenha que ser definida a variável de ambiente `PGPORT`. A regra básica é esta: quando se tenta iniciar um programa aplicativo e este informa que não está conseguindo conectar com o banco de dados, deve ser consultado o administrador do servidor ou, caso seja você mesmo, a documentação, para ter certeza que o ambiente está configurado de maneira apropriada. Caso não tenha entendido o parágrafo anterior então, por favor, leia a próxima seção.

1.2. Fundamentos da arquitetura

Antes de prosseguir, é necessário conhecer a arquitetura de sistema básica do PostgreSQL. Compreender como as partes do PostgreSQL interagem torna este capítulo mais claro.

No jargão de banco de dados, o PostgreSQL utiliza o modelo cliente-servidor. Uma sessão do PostgreSQL consiste nos seguintes processos (programas) cooperando entre si:

- Um processo servidor, que gerencia os arquivos de banco de dados, recebe conexões dos aplicativos cliente com o banco de dados, e executa ações no banco de dados em nome dos clientes. O programa servidor de banco de dados se chama `postmaster`.
- O aplicativo cliente do usuário (`frontend`) que deseja executar operações de banco de dados. Os aplicativos cliente podem ter naturezas muito diversas: o cliente pode ser uma ferramenta no modo caractere, um aplicativo gráfico, um servidor Web que acessa o banco de dados para mostrar páginas Web, ou uma ferramenta especializada para manutenção do banco de dados. Alguns aplicativos cliente são fornecidos na distribuição do PostgreSQL, sendo a maioria desenvolvido pelos usuários. [2]

Como é típico em aplicações cliente-servidor, o cliente e o servidor podem estar em hospedeiros diferentes. Neste caso se comunicam através de uma conexão de rede TCP/IP. Deve-se ter isto em mente, porque arquivos que podem ser acessados na máquina cliente podem não ser acessíveis pela máquina servidora (ou somente podem ser acessados usando um nome de arquivo diferente).

O servidor PostgreSQL pode tratar várias conexões simultâneas de clientes. Para esta finalidade é iniciado um novo processo (`fork` [3]) para cada conexão. Deste ponto em diante, o cliente e o novo processo servidor se comunicam sem intervenção do processo `postmaster` original. Portanto, o `postmaster` está sempre executando aguardando por novas conexões dos clientes, enquanto os clientes e seus processos servidor associados surgem e desaparecem (obviamente tudo isso é invisível para o usuário, sendo mencionado somente para ficar completo).

1.3. Criação de banco de dados

O primeiro teste para verificar se é possível acessar o servidor de banco de dados é tentar criar um banco de dados. Um servidor PostgreSQL pode gerenciar muitos bancos de dados. Normalmente é utilizado um banco de dados em separado para cada projeto ou para cada usuário.

Possivelmente, o administrador já criou um banco de dados para seu uso. Ele deve ter dito qual é o nome do seu banco de dados. Neste caso esta etapa pode ser omitida, indo-se direto para a próxima seção.

Para criar um novo banco de dados, chamado `meu_bd` neste exemplo, deve ser utilizado o comando:

```
$ createdb meu_bd
```

Que deve produzir a seguinte resposta:

```
CREATE DATABASE
```

Se esta resposta for mostrada então esta etapa foi bem sucedida, podendo-se pular o restante da seção .

Se for mostrada uma mensagem semelhante a

```
createdb: comando não encontrado
```

então o PostgreSQL não foi instalado da maneira correta, ou não foi instalado, ou o caminho de procura não foi definido corretamente. Tente executar o comando utilizando o caminho absoluto:

```
$ /usr/local/pgsql/bin/createdb meu_bd
```

O caminho na sua máquina pode ser diferente. [4] Fale com o administrador, ou verifique novamente as instruções de instalação para corrigir a situação.

Outra resposta pode ser esta: [5]

```
createdb: could not connect to database template1: could not connect to server:
No such file or directory
        Is the server running locally and accepting
```

```
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

-- Tradução da mensagem (N. do T.)

```
createdb: não foi possível conectar ao banco de dados template1
        : não foi possível conectar ao servidor:
Arquivo ou diretório inexistente
        O servidor está executando localmente e aceitando
        conexões no soquete do domínio Unix "/tmp/.s.PGSQL.5432"?
```

Significando que o servidor não foi inicializado, ou que não foi inicializado onde o `createdb` esperava que fosse. Novamente, verifique as instruções de instalação ou consulte o administrador.

Outra resposta pode ser esta:

```
createdb: could not connect to database template1:
FATAL: user "joel" does not exist
```

-- Tradução da mensagem (N. do T.)

```
createdb: não foi possível conectar ao banco de dados template1:
FATAL: o usuário "joel" não existe
```

onde é mencionado o seu próprio nome de login. Isto vai acontecer se o administrador não tiver criado uma conta de usuário no PostgreSQL para seu uso (As contas de usuário do PostgreSQL são distintas das contas de usuário do sistema operacional). Se você for o administrador, obtenha ajuda para criar contas no Capítulo 17. Será necessário se tornar o usuário do sistema operacional que instalou o PostgreSQL (geralmente `postgres`) para criar a primeira conta de usuário. Também pode ter sido atribuído para você um nome de usuário do PostgreSQL diferente do nome de usuário do sistema operacional; neste caso, é necessário utilizar a chave `-U`, ou definir a variável de ambiente `PGUSER`, para especificar o nome de usuário do PostgreSQL.

Caso se possua uma conta de usuário, mas esta conta não possua o privilégio necessário para criar bancos de dados, será exibida a seguinte mensagem:

```
createdb: database creation failed:
ERROR: permission denied to create database
```

-- Tradução da mensagem (N. do T.)

```
createdb: a criação do banco de dados falhou:
ERRO: negada a permissão para criar banco de dados
```

Nem todo usuário possui autorização para criar bancos de dados. Se o PostgreSQL não permitir criar o banco de dados, então o administrador deve conceder permissão para você criar bancos de dados. Consulte o administrador caso isto ocorra. Caso tenha instalado o PostgreSQL por si próprio, então conecte usando a mesma conta de usuário utilizada para inicializar o servidor, para as finalidades deste tutorial. [6]

Também podem ser criados bancos de dados com outros nomes. O PostgreSQL permite a criação de qualquer número de bancos de dados em uma instalação. Os nomes dos bancos de dados devem ter o primeiro caractere alfabético, sendo limitados a um comprimento de 63 caracteres. Uma escolha conveniente é criar o banco de dados com o mesmo nome do usuário corrente. Muitas ferramentas assumem este nome de banco de dados como sendo o nome padrão, evitando a necessidade de digitá-lo. Para criar este banco de dados deve ser digitado simplesmente:

```
$ createdb
```

Caso não deseje mais utilizar o seu banco de dados, pode removê-lo. Por exemplo, se você for o dono (criador) do banco de dados `meu_bd`, poderá removê-lo utilizando o seguinte comando:

```
$ dropdb meu_bd
```

Para este comando o nome da conta não é utilizado como nome padrão do banco de dados: o nome sempre deve ser especificado. Esta ação remove fisicamente todos os arquivos associados ao banco de dados não podendo ser desfeita, portanto esta operação somente deve ser feita após um longo período de reflexão.

Podem ser encontradas informações adicionais sobre os comandos `createdb` e `dropdb` em createdb e dropdb, respectivamente.

1.4. Acesso a banco de dados

Após o banco de dados ter sido criado, este pode ser acessado pela:

- Execução do programa de terminal interativo do PostgreSQL chamado *psql*, que permite entrar, editar e executar comandos SQL interativamente.
- Utilização de uma ferramenta cliente gráfica existente como o PgAccess, ou de um pacote de automação de escritórios com suporte a ODBC para criar e manusear bancos de dados. Estas possibilidades não estão descritas neste tutorial.
- Criação de aplicativos personalizados, usando um dos vários vínculos com linguagens disponíveis. Estas possibilidades são mostradas mais detalhadamente na Parte IV.

Você provavelmente vai desejar ativar o `psql` para executar os exemplos deste tutorial. O `psql` pode ser ativado para usar o banco de dados `meu_bd` digitando o comando:

```
$ psql meu_bd
```

Se o nome do banco de dados for omitido, então será usado o nome padrão igual ao nome da conta do usuário. Isto já foi visto na seção anterior.

O `psql` saúda o usuário com a seguinte mensagem:

```
Welcome to psql 8.0.0, the PostgreSQL interactive terminal.


Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit


meu_bd=>
```

```
-- Tradução da mensagem (N. do T.)

Bem-vindo ao psql 8.0.0, o terminal interativo do PostgreSQL.


Digite:  \copyright para mostrar a licença da distribuição
         \h para ajuda nos comandos SQL
         \? para ajuda nos comandos de contrabarra internos
         \g ou finalizar com ponto-e-vírgula para executar o comando
         \q para sair


meu_bd=>
```

A última linha também pode ser

```
meu_bd=#
```

significando que o usuário é um superusuário do banco de dados, acontecendo geralmente quando se instala o PostgreSQL por si próprio. Ser um superusuário significa não estar sujeito a controles de acesso. Para as finalidades deste tutorial isto não tem importância.

Caso aconteçam problemas ao inicializar o `psql`, então retorne à seção anterior. Os diagnósticos do `psql` e do `createdb` são semelhantes, e se um funcionou o outro deve funcionar também.

A última linha exibida pelo `psql` é o `prompt`, indicando que o `psql` está lhe aguardando, e que você pode digitar comandos SQL dentro do espaço de trabalho mantido pelo `psql`. Tente estes comandos:

```
meu_bd=> SELECT version();

                           version
------------------------------------------------------------
 PostgreSQL 8.0.0 on i586-pc-linux-gnu, compiled by GCC 2.96
(1 linha)

meu_bd=> SELECT current_date;

    date
------------
 2005-05-17
(1 linha)

meu_bd=> SELECT 2 + 2;

 ?column?
----------
        4
(1 linha)
```

O programa `psql` possui vários comandos internos que não são comandos SQL. Eles começam pelo caractere de contrabarra, "\". Alguns destes comandos são mostrados na mensagem de boas vindas. Por exemplo, pode ser obtida ajuda sobre a sintaxe de vários comandos SQL do PostgreSQL digitando:

```
meu_bd=> \h
```

Para sair do `psql` digite

```
meu_bd=> \q
```

e o `psql` terminará retornando para o interpretador de comandos (para conhecer outros comandos internos digite `\?` no `prompt` do `psql`). Todas as funcionalidades do `psql` estão documentadas em psql. Se o PostgreSQL tiver sido instalado corretamente, também pode-se digitar `man psql` na linha de comando do sistema operacional para ver a documentação. Neste tutorial não utilizaremos estas funcionalidades explicitamente, mas use por si próprio quando julgar adequado.

Notas

1. `distribuição` — Uma árvore de código fonte de `software` empacotada para distribuição; Desde cerca de 1996 a utilização não qualificada deste termo geralmente implica numa "distribuição Linux". A forma

curta “distro” geralmente é utilizada neste sentido. The Jargon File (http://www.catb.org/~esr/jargon/html/D/distribution.html) (N. do T.)

2. Também pode ser utilizada a ferramenta de administração do PostgreSQL baseada na Web phpPgAdmin (http://phppgadmin.sourceforge.net/). (N. do T.)

3. `fork` — Para criar um novo processo, o processo copia a si próprio através da chamada de sistema `fork`. O `fork` cria uma cópia do processo original que é em grande parte idêntica à ancestral. O novo processo possui um PID (identificador de processo) próprio, e suas próprias informações de contabilização. O `fork` possui a propriedade única de retornar dois valores diferentes. Do ponto de vista do filho retorna zero. Por outro lado, para o pai é retornado o PID do filho recém criado. Uma vez que fora isso os dois processos são idênticos, ambos precisam examinar o valor retornado para descobrir o papel a ser desempenhado. Linux Administration Handbook, Evi Nemeth e outros, Prentice Hall, 2002. (N. do T.)

4. `/usr/bin/createdb` no RedHat, Fedora, Mandrake e Debian. (N. do T.)

5. soquete — As chamadas de sistema para estabelecer a conexão são um tanto diferentes para o cliente e para o servidor, mas ambas envolvem basicamente a construção de um soquete. O soquete é uma extremidade do canal de comunicação entre processos. Cada um dos dois processos estabelece seu próprio soquete. Para criar o soquete, o programa precisa especificar o domínio de endereço e o tipo de soquete. Dois processos podem se comunicar somente se seus soquetes são do mesmo tipo e do mesmo domínio. Existem dois domínios de endereço amplamente utilizados: o domínio Unix, no qual dois processos compartilham um arquivo do sistema em comum para se comunicar; e o domínio Internet, no qual dois processos executando em dois hospedeiros quaisquer na Internet se comunicam. Cada um destes domínios possui seu próprio formato de endereço. O endereço de soquete do domínio Unix é uma cadeia de caracteres, que é basicamente uma entrada no sistema de arquivos. O endereço de soquete do domínio Internet consiste no endereço de Internet da máquina hospedeira (todo computador na Internet possui um endereço único de 32 bits, geralmente chamado de endereço de IP). Adicionalmente, no domínio Internet, cada soquete necessita do número da porta no hospedeiro. Sockets Tutorial (http://people.cs.uchicago.edu/%7Emark/51081/labs/LAB6/sock.html) (N. do T.)

6. Uma explicação do motivo pelo qual isto funciona: Os nomes de usuário do PostgreSQL são distintos das contas de usuário do sistema operacional. Ao estabelecer a conexão com um banco de dados, pode ser escolhido o nome do usuário do PostgreSQL com o qual se deseja fazer a conexão; Se isto não for feito, o padrão é utilizar um nome igual ao da conta atual do sistema operacional. Como isto ocorre, sempre existirá uma conta de usuário do PostgreSQL que possui o nome igual ao do usuário do sistema operacional que inicializou o servidor; acontece, também, que este usuário sempre tem permissão para criar banco de dados. Em vez de conectar como este usuário, pode ser especificada a opção `-U` em todos os aplicativos para escolher o nome do usuário do PostgreSQL com o qual se deseja conectar.

Capítulo 2. A linguagem SQL

2.1. Introdução

Este capítulo fornece uma visão geral sobre como utilizar a linguagem SQL para realizar operações simples. O propósito deste tutorial é apenas fazer uma introdução e, de forma alguma, ser um tutorial completo sobre a linguagem SQL. Existem muitos livros escritos sobre a linguagem SQL, incluindo *Understanding the New SQL* e *A Guide to the SQL Standard*. É preciso estar ciente que algumas funcionalidades da linguagem SQL do PostgreSQL são extensões ao padrão.

Nos exemplos a seguir supõe-se que tenha sido criado o banco de dados chamado `meu_bd`, conforme descrito no capítulo anterior, e que o psql esteja ativo.

Os exemplos presentes neste manual também podem ser encontrados na distribuição do código fonte do PostgreSQL, no diretório `src/tutorial/`. [1] Para usar estes arquivos, primeiro deve-se tornar o diretório `src/tutorial/` o diretório corrente, e depois executar o utilitário make, conforme mostrado abaixo:

```
$ cd ..../src/tutorial
$ make
```

Este procedimento cria os scripts e compila os arquivos C contendo as funções e tipos definidos pelo usuário (Deve ser utilizado o make do GNU neste procedimento, que pode ter um nome diferente no sistema sendo utilizado, geralmente gmake). [2] Depois disso, para iniciar o tutorial faça o seguinte:

```
$ cd ..../src/tutorial
$ psql -s meu_bd
...

meu_bd=> \i basics.sql
```

O comando `\i` lê os comandos no arquivo especificado. A opção `-s` ativa o modo passo a passo, que faz uma pausa antes de enviar cada comando para o servidor. Os comandos utilizados nesta seção estão no arquivo basics.sql (./basics.sql).

2.2. Conceitos

O PostgreSQL é um *sistema de gerenciamento de banco de dados relacional* (SGBDR). Isto significa que é um sistema para gerenciar dados armazenados em *relações*. Relação é, essencialmente, um termo matemático para *tabela*. A noção de armazenar dados em tabelas é tão trivial hoje em dia que pode parecer totalmente óbvio, mas existem várias outras formas de organizar bancos de dados. Arquivos e diretórios em sistemas operacionais tipo Unix são um exemplo de banco de dados hierárquico. Um desenvolvimento mais moderno são os bancos de dados orientados a objeto.

Cada tabela é uma coleção nomeada de *linhas*. Todas as linhas de uma determinada tabela possuem o mesmo conjunto de *colunas* nomeadas, e cada coluna é de um tipo de dado específico. Enquanto as colunas possuem uma ordem fixa nas linhas, é importante lembrar que o SQL não garante a ordem das linhas dentro de uma tabela (embora as linhas possam ser explicitamente ordenadas para a exibição).

As tabelas são agrupadas em bancos de dados, e uma coleção de bancos de dados gerenciados por uma única instância do servidor PostgreSQL forma um *agrupamento* de bancos de dados.

2.3. Criação de tabelas

Pode-se criar uma tabela especificando o seu nome juntamente com os nomes das colunas e seus tipos de dado:

```
CREATE TABLE clima (
    cidade          varchar(80),
    temp_min        int,            -- temperatura mínima
    temp_max        int,            -- temperatura máxima
    prcp            real,           -- precipitação
    data            date
);
```

Este comando pode ser digitado no `psql` com quebras de linha. O `psql` reconhece que o comando só termina quando é encontrado o ponto-e-vírgula.

Espaços em branco (ou seja, espaços, tabulações e novas linhas) podem ser utilizados livremente nos comandos SQL. Isto significa que o comando pode ser digitado com um alinhamento diferente do mostrado acima, ou mesmo tudo em uma única linha. Dois hífens ("`--`") iniciam um comentário; tudo que vem depois é ignorado até o final da linha. A linguagem SQL não diferencia letras maiúsculas e minúsculas nas palavras chave e nos identificadores, a não ser que os identificadores sejam colocados entre aspas (") para preservar letras maiúsculas e minúsculas, o que não foi feito acima.

No comando, `varchar(80)` especifica um tipo de dado que pode armazenar cadeias de caracteres arbitrárias com comprimento até 80 caracteres; `int` é o tipo inteiro normal; `real` é o tipo para armazenar números de ponto flutuante de precisão simples; `date` é o tipo para armazenar data e hora (a coluna do tipo `date` pode se chamar `date`, o que tanto pode ser conveniente quanto pode causar confusão).

O PostgreSQL suporta os tipos SQL padrão `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp` e `interval`, assim como outros tipos de utilidade geral, e um conjunto abrangente de tipos geométricos. O PostgreSQL pode ser personalizado com um número arbitrário de tipos definidos pelo usuário. Como conseqüência, sintaticamente os nomes dos tipos não são palavras chave, exceto onde for requerido para suportar casos especiais do padrão SQL.

No segundo exemplo são armazenadas cidades e suas localizações geográficas associadas:

```
CREATE TABLE cidades (
    nome            varchar(80),
    localizacao     point
);
```

O tipo `point` é um exemplo de tipo de dado específico do PostgreSQL.

Para terminar deve ser mencionado que, quando a tabela não é mais necessária, ou se deseja recriá-la de uma forma diferente, é possível removê-la por meio do comando:

```
DROP TABLE nome_da_tabela;
```

2.4. Inserção de linhas em tabelas

É utilizado o comando `INSERT` para inserir linhas nas tabelas:

```
INSERT INTO clima VALUES ('São Francisco', 46, 50, 0.25, '1994-11-27');
```

Repare que todos os tipos de dado possuem formato de entrada de dados bastante óbvios. As constantes, que não são apenas valores numéricos, geralmente devem estar entre apóstrofos (`'`), como no exemplo acima. O tipo `date` é, na verdade, muito flexível em relação aos dados que aceita, mas para este tutorial vamos nos fixar no formato sem ambigüidade mostrado acima.

O tipo `point` requer um par de coordenadas como entrada, como mostrado abaixo:

```
INSERT INTO cidades VALUES ('São Francisco', '(-194.0, 53.0)');
```

A sintaxe usada até agora requer que seja lembrada a ordem das colunas. Uma sintaxe alternativa permite declarar as colunas explicitamente:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, data)
    VALUES ('São Francisco', 43, 57, 0.0, '1994-11-29');
```

Se for desejado, pode-se declarar as colunas em uma ordem diferente, e pode-se, também, omitir algumas colunas. Por exemplo, se a precipitação não for conhecida:

```
INSERT INTO clima (data, cidade, temp_max, temp_min)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Muitos desenvolvedores consideram declarar explicitamente as colunas um estilo melhor que confiar na ordem implícita.

Por favor, entre todos os comando mostrados acima para ter alguns dados para trabalhar nas próximas seções.

Também pode ser utilizado o comando `COPY` para carregar uma grande quantidade de dados a partir de arquivos texto puro. Geralmente é mais rápido, porque o comando `COPY` é otimizado para esta finalidade, embora possua menos flexibilidade que o comando `INSERT`. Para servir de exemplo:

```
COPY clima FROM '/home/user/clima.txt';
```

O arquivo contendo os dados deve poder ser acessado pelo servidor e não pelo cliente, porque o servidor lê o arquivo diretamente. Podem ser obtidas mais informações sobre o comando COPY em COPY.

2.5. Consultar tabelas

Para trazer os dados de uma tabela, a tabela deve ser *consultada*. Para esta finalidade é utilizado o comando SELECT do SQL. Este comando é dividido em *lista de seleção* (a parte que especifica as colunas a serem trazidas), *lista de tabelas* (a parte que especifica as tabelas de onde os dados vão ser trazidos), e uma *qualificação opcional* (a parte onde são especificadas as restrições). Por exemplo, para trazer todas as linhas da tabela clima digite:

```
SELECT * FROM clima;
```

(aqui * é uma forma abreviada de "todas as colunas").[3] Seriam obtidos os mesmos resultados usando:

```
SELECT cidade, temp_min, temp_max, prcp, data FROM clima;
```

A saída deve ser:

```
     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
 São Francisco   |       43 |       57 |    0 | 1994-11-29
 Hayward         |       37 |       54 |      | 1994-11-29
(3 linhas)
```

Na lista de seleção podem ser especificadas expressões, e não apenas referências a colunas. Por exemplo, pode ser escrito

```
SELECT cidade, (temp_max+temp_min)/2 AS temp_media, data FROM clima;
```

devendo produzir:

```
     cidade      | temp_media |    data
-----------------+------------+------------
 São Francisco   |         48 | 1994-11-27
 São Francisco   |         50 | 1994-11-29
 Hayward         |         45 | 1994-11-29
(3 linhas)
```

Perceba que a cláusula AS foi utilizada para mudar o nome da coluna de saída (a cláusula AS é opcional).

A consulta pode ser "qualificada", adicionando a cláusula WHERE para especificar as linhas desejadas. A cláusula WHERE contém expressões booleanas (valor verdade), e somente são retornadas as linhas para as quais o valor da expressão booleana for verdade. São permitidos os operadores booleanos usuais (AND, OR e NOT) na qualificação. Por exemplo, o comando abaixo retorna os registros do clima de São Francisco nos dias de chuva:

```
SELECT * FROM clima
    WHERE cidade = 'São Francisco' AND prcp > 0.0;
```

Resultado:

```
     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
(1 linha)
```

Pode ser solicitado que os resultados da consulta sejam retornados em uma determinada ordem:

```
SELECT * FROM clima
    ORDER BY cidade;
```

```
     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 Hayward         |       37 |       54 |      | 1994-11-29
 São Francisco   |       43 |       57 |    0 | 1994-11-29
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
```

Neste exemplo a ordem de classificação não está totalmente especificada e, portanto, as linhas de São Francisco podem retornar em qualquer ordem. Mas sempre seriam obtidos os resultados mostrados acima se fosse executado:

```
SELECT * FROM clima
    ORDER BY cidade, temp_min;
```

Pode ser solicitado que as linhas duplicadas sejam removidas do resultado da consulta: [4]

```
SELECT DISTINCT cidade
    FROM clima;
```

```
    cidade
---------------
 Hayward
 São Francisco
(2 linhas)
```

Novamente, neste exemplo a ordem das linhas pode variar. Pode-se garantir resultados consistentes utilizando `DISTINCT` e `ORDER BY` juntos: [5] [6]

```
SELECT DISTINCT cidade
    FROM clima
    ORDER BY cidade;
```

2.6. Junções entre tabelas

Até agora as consultas somente acessaram uma tabela de cada vez. As consultas podem acessar várias tabelas de uma vez, ou acessar a mesma tabela de uma maneira que várias linhas da tabela sejam processadas ao mesmo tempo. A consulta que acessa várias linhas da mesma tabela, ou de tabelas diferentes, de uma vez, é chamada de consulta de *junção*. Como exemplo, suponha que se queira listar todas as linhas de clima junto com a localização da cidade associada. Para se fazer isto, é necessário comparar a coluna cidade de cada linha da tabela clima com a coluna nome de todas as linhas da tabela cidades, e selecionar os pares de linha onde estes valores são correspondentes.

> **Nota:** Este é apenas um modelo conceitual, a junção geralmente é realizada de uma maneira mais eficiente que comparar de verdade cada par de linhas possível, mas isto não é visível para o usuário.

Esta operação pode ser efetuada por meio da seguinte consulta:

```
SELECT *
    FROM clima, cidades
    WHERE cidade = nome;
```

```
     cidade      | temp_min | temp_max | prcp |    data    |     nome      | localizacao
-----------------+----------+----------+------+------------+---------------+-------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27 | São Francisco | (-194,53)
 São Francisco   |       43 |       57 |    0 | 1994-11-29 | São Francisco | (-194,53)
(2 linhas)
```

Duas coisas devem ser observadas no resultado produzido:

- Não existe nenhuma linha para a cidade Hayward. Isto acontece porque não existe entrada correspondente na tabela `cidades` para Hayward, e a junção ignora as linhas da tabela clima sem correspondência. Veremos em breve como isto pode ser mudado.
- Existem duas colunas contendo o nome da cidade, o que está correto porque a lista de colunas das tabelas `clima` e `cidades` estão concatenadas. Na prática isto não é desejado, sendo preferível, portanto, escrever a lista das colunas de saída explicitamente em vez de utilizar o *:

```
SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
    WHERE cidade = nome;
```

Exercício: Descobrir a semântica desta consulta quando a cláusula `WHERE` é omitida.

Como todas as colunas possuem nomes diferentes, o analisador encontra automaticamente a tabela que a coluna pertence, mas é um bom estilo qualificar completamente os nomes das colunas nas consultas de junção:

```
SELECT clima.cidade, clima.temp_min, clima.temp_max,
       clima.prcp, clima.data, cidades.localizacao
    FROM clima, cidades
    WHERE cidades.nome = clima.cidade;
```

As consultas de junção do tipo visto até agora também poderiam ser escritas da seguinte forma alternativa:

```
SELECT *
    FROM clima INNER JOIN cidades ON (clima.cidade = cidades.nome);
```

A utilização desta sintaxe não é tão comum quanto a usada acima, mas é mostrada para ajudar a entender os próximos tópicos.

Agora vamos descobrir como se faz para obter as linhas de Hayward. Desejamos o seguinte: que a consulta varra a tabela `clima` e, para cada uma de suas linhas, encontre a linha correspondente na tabela `cidades`. Se não for encontrada nenhuma linha correspondente, desejamos que sejam colocados "valores vazios" nas colunas da tabela `cidades`. Este tipo de consulta é chamada de *junção externa* (`outer join`). As consultas vistas até agora são junções internas (`inner join`). O comando então fica assim:

```
SELECT *
    FROM clima LEFT OUTER JOIN cidades ON (clima.cidade = cidades.nome);
```

```
     cidade      | temp_min | temp_max | prcp |    data    |     nome      | localizacao
-----------------+----------+----------+------+------------+---------------+------------
 Hayward         |       37 |       54 |      | 1994-11-29 |               |
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27 | São Francisco | (-194,53)
 São Francisco   |       43 |       57 |    0 | 1994-11-29 | São Francisco | (-194,53)
(3 linhas)
```

Esta consulta é chamada de *junção externa esquerda* (`left outer join`), porque a tabela mencionada à esquerda do operador de junção terá cada uma de suas linhas aparecendo na saída pelo menos uma vez, enquanto a tabela à direita terá somente as linhas correspondendo a alguma linha da tabela à esquerda aparecendo na saída. Ao listar uma linha da tabela à esquerda, para a qual não existe nenhuma linha correspondente na tabela à direita, são colocados valores vazios (`null`) nas colunas da tabela à direita.

Exercício: Existem também a junção externa direita (`right outer join`) e a junção externa completa (`full outer join`). Tente descobrir o que fazem.

Também é possível fazer a junção da tabela consigo mesma. Isto é chamado de *autojunção* (`self join`). Como exemplo, suponha que desejamos descobrir todas as linhas de clima que estão no intervalo de temperatura de outros registros de clima. Para isso é necessário comparar as colunas `temp_min` e `temp_max` de cada registro de `clima` com as colunas `temp_min` e `temp_max` de todos os outros registros da tabela `clima`, o que pode ser feito utilizando a seguinte consulta:

```
SELECT C1.cidade, C1.temp_min AS menor, C1.temp_max AS maior,
    C2.cidade, C2.temp_min AS menor, C2.temp_max AS maior
    FROM clima C1, clima C2
    WHERE C1.temp_min < C2.temp_min
    AND C1.temp_max > C2.temp_max;
```

```
     cidade      | menor | maior |     cidade    | menor | maior
-----------------+-------+-------+---------------+-------+-------
 São Francisco   |    43 |    57 | São Francisco |    46 |    50
 Hayward         |    37 |    54 | São Francisco |    46 |    50
(2 linhas)
```

A tabela clima teve seu nome mudado para `C1` e `C2`, para permitir distinguir o lado esquerdo do lado direito da junção. Estes tipos de "aliases" também podem ser utilizados em outras consultas para reduzir a digitação como, por exemplo:

```
SELECT *
    FROM clima w, cidades c
    WHERE w.cidade = c.nome;
```

Será vista esta forma de abreviar com bastante freqüência.

2.7. Funções de agregação

Como a maioria dos produtos de banco de dados relacional, o PostgreSQL suporta funções de agregação. Uma função de agregação computa um único resultado para várias linhas de entrada. Por exemplo, existem funções de agregação para contar (`count`), somar (`sum`), calcular a média (`avg`), o valor máximo (`max`) e o valor mínimo (`min`) para um conjunto de linhas.

Para servir de exemplo, é possível encontrar a maior temperatura mínima ocorrida em qualquer lugar usando

```
SELECT max(temp_min) FROM clima;
```

```
 max
-----
  46
(1 linha)
```

Se for desejado saber a cidade (ou cidades) onde esta temperatura ocorreu pode-se tentar usar

```
SELECT cidade FROM clima WHERE temp_min = max(temp_min);      ERRADO
```

mas não vai funcionar, porque a função de agregação `max` não pode ser usada na cláusula `WHERE` (Esta restrição existe porque a cláusula `WHERE` determina as linhas que vão passar para o estágio de agregação e, portanto, precisa ser avaliada antes das funções de agregação serem computadas). Entretanto, como é geralmente o caso, a consulta pode ser reformulada para obter o resultado pretendido, o que será feito por meio de uma *subconsulta*:

```
SELECT cidade FROM clima
    WHERE temp_min = (SELECT max(temp_min) FROM clima);
```

```
     cidade
---------------
 São Francisco
(1 linha)
```

Isto está correto porque a subconsulta é uma ação independente, que calcula sua agregação em separado do que está acontecendo na consulta externa.

As agregações também são muito úteis em combinação com a cláusula `GROUP BY`. Por exemplo, pode ser obtida a maior temperatura mínima observada em cada cidade usando

```
SELECT cidade, max(temp_min)
    FROM clima
    GROUP BY cidade;
```

```
     cidade      | max
-----------------+-----
 Hayward         |  37
 São Francisco   |  46
(2 linhas)
```

produzindo uma linha de saída para cada cidade. Cada resultado da agregação é computado sobre as linhas da tabela correspondendo a uma cidade. As linhas agrupadas podem ser filtradas utilizando a cláusula `HAVING`

```
SELECT cidade, max(temp_min)
    FROM clima
    GROUP BY cidade
    HAVING max(temp_min) < 40;
```

```
  cidade   | max
-----------+-----
 Hayward   |  37
(1 linha)
```

que mostra os mesmos resultados, mas apenas para as cidades que possuem todos os valores de `temp_min` abaixo de 40. Para concluir, se desejarmos somente as cidades com nome começando pela letra “S” podemos escrever:

```
SELECT cidade, max(temp_min)
    FROM clima
    WHERE cidade LIKE 'S%'Œ
    GROUP BY cidade
    HAVING max(temp_min) < 40;
```

Œ O operador `LIKE` faz correspondência com padrão, sendo explicado na Seção 9.7.

É importante compreender a interação entre as agregações e as cláusulas `WHERE` e `HAVING` do SQL. A diferença fundamental entre `WHERE` e `HAVING` é esta: `WHERE` seleciona as linhas de entrada antes dos grupos e agregações serem computados (portanto, controla quais linhas irão para o computo da agregação), enquanto `HAVING` seleciona linhas de grupo após os grupos e agregações serem computados. Portanto, a cláusula `WHERE` não pode conter funções de agregação; não faz sentido tentar utilizar uma agregação para determinar quais linhas serão a entrada da agregação. Por outro lado, a cláusula `HAVING` sempre contém funções de agregação (A rigor, é permitido escrever uma cláusula `HAVING` que não possua agregação, mas é desperdício: A mesma condição poderia ser utilizada de forma mais eficiente no estágio do `WHERE`). [7]

No exemplo anterior, a restrição do nome da cidade pode ser aplicada na cláusula `WHERE`, porque não necessita de nenhuma agregação, sendo mais eficiente que colocar a restrição na cláusula `HAVING`, porque evita realizar os procedimentos de agrupamento e agregação em todas as linhas que não atendem a cláusula `WHERE`.

2.8. Atualizações

As linhas existentes podem ser atualizadas utilizando o comando `UPDATE`. Suponha que foi descoberto que as leituras de temperatura estão todas mais altas 2 graus após 28 de novembro de 1994. Os dados podem ser atualizados da seguinte maneira:

```
UPDATE clima
    SET temp_max = temp_max - 2,  temp_min = temp_min - 2
    WHERE data > '1994-11-28';
```

Agora vejamos o novo estado dos dados:

```
SELECT * FROM clima;
```

```
     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
 São Francisco   |       41 |       55 |    0 | 1994-11-29
 Hayward         |       35 |       52 |      | 1994-11-29
(3 linhas)
```

2.9. Exclusões

As linhas podem ser removidas da tabela através do comando DELETE. Suponha que não estamos mais interessados nos registros do clima em Hayward. Então precisamos excluir estas linhas da tabela.

```
DELETE FROM clima WHERE cidade = 'Hayward';
```

Todos os registros de clima pertencentes a Hayward são removidos.

```
SELECT * FROM clima;
```

```
    cidade     | temp_min | temp_max | prcp |    data
---------------+----------+----------+------+------------
 São Francisco |       46 |       50 | 0.25 | 1994-11-27
 São Francisco |       41 |       55 |    0 | 1994-11-29
(2 linhas)
```

Deve-se tomar cuidado com comandos na forma:

```
DELETE FROM nome_da_tabela;
```

Sem uma qualificação, o comando DELETE remove *todas* as linhas da tabela, deixando-a vazia. O sistema não solicita confirmação antes de realizar esta operação!

Notas

1. Os arquivos basics.sql (./basics.sql) e advanced.sql (./advanced.sql) foram traduzidos e colocados como `links` nesta tradução. Para usá-los basta salvar os arquivos em disco, abrir o interpretador de comandos, tornar o diretório onde os arquivos foram salvos o diretório corrente, executar na linha de comando `psql -s meu_bd` e usar o comando `\i basics.sql` ou `\i advanced.sql` para executar o arquivo, como mostrado neste capítulo. (N. do T.)
2. Use o comando make --version para saber se o make utilizado é o make do GNU. (N. do T.)
3. Embora o SELECT * seja útil para consultas rápidas, geralmente é considerado um estilo ruim para código em produção, uma vez que a adição de uma coluna à tabela mudaria os resultados.
4. Para o Oracle 9i as palavras chave UNIQUE e DISTINCT são sinônimos, podendo ser usada qualquer uma das duas no comando SELECT. (N. do T.)
5. Em alguns sistemas de banco de dados, incluindo as versões antigas do PostgreSQL, a implementação do DISTINCT ordena automaticamente as linhas e, por isso, o ORDER BY é redundante. Mas isto não é requerido pelo padrão SQL, e o PostgreSQL corrente não garante que DISTINCT faça com que as linhas sejam ordenadas.

6. Oracle 9i — Se for especificado o operador `DISTINCT` no comando `SELECT`, então a cláusula `ORDER BY` não pode fazer referência a colunas que não aparecem na lista de seleção. (Foi observado que esta restrição também se aplica ao PostgreSQL, ao DB2 e ao SQL Server). (N. do T.)
7. Consulte o Exemplo 7-1 (N. do T.)

Capítulo 3. Funcionalidades avançadas

3.1. Introdução

Nos capítulos anteriores foi descrita a utilização básica da linguagem SQL para armazenar e acessar dados no PostgreSQL. Agora serão mostradas algumas funcionalidades mais avançadas da linguagem SQL que simplificam a gerência, e evitam a perda e a corrupção dos dados. No final serão vistas algumas extensões do PostgreSQL.

Em certas ocasiões este capítulo faz referência aos exemplos encontrados no Capítulo 2 para modificá-los ou melhorá-los, portanto recomenda-se que este capítulo já tenha sido lido. Alguns exemplos do presente capítulo também se encontram no arquivo advanced.sql (./advanced.sql) no diretório do tutorial. Este arquivo também contém dados dos exemplos a serem carregados, que não serão repetidos aqui (consulte a Seção 2.1 para saber como usar este arquivo).

3.2. Visões

Reveja as consultas na Seção 2.6. Supondo que a consulta combinando os registros de clima e de localização das cidades seja de particular interesse para um aplicativo, mas que não se deseja digitar esta consulta toda vez que for necessária, então é possível criar uma *visão* baseada na consulta, atribuindo um nome a esta consulta pelo qual será possível referenciá-la como se fosse uma tabela comum.

```
CREATE VIEW minha_visao AS
    SELECT cidade, temp_min, temp_max, prcp, data, localizacao
        FROM clima, cidades
        WHERE cidade = nome;

SELECT * FROM minha_visao;
```

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL. As visões permitem encapsular, atrás de interfaces que não mudam, os detalhes da estrutura das tabelas, que podem mudar na medida em que os aplicativos evoluem.

As visões podem ser utilizadas em praticamente todos os lugares onde uma tabela real pode ser utilizada. Construir visões baseadas em visões não é raro.

3.3. Chaves estrangeiras

Reveja as tabelas `clima` e `cidades` no Capítulo 2. Considere o seguinte problema: Desejamos ter certeza que não serão inseridas linhas na tabela `clima` sem que haja um registro correspondente na tabela `cidades`. Isto é chamado de manter a *integridade referencial* dos dados. Em sistemas de banco de dados muito simples poderia ser implementado (caso fosse) olhando primeiro a tabela `cidades` para verificar se existe a linha correspondente e, depois, inserir ou rejeitar a nova linha de `clima`. Esta abordagem possui vários problemas, e é muito inconveniente, por isso o PostgreSQL pode realizar esta operação por você.

A nova declaração das tabelas ficaria assim:

```
CREATE TABLE cidades (
        cidade      varchar(80) primary key,
        localizacao point
);
```

```
CREATE TABLE clima (
        cidade     varchar(80) references cidades(cidade),
        temp_min   int,
        temp_max   int,
        prcp       real,
        data       date
);
```

Agora, ao se tentar inserir uma linha inválida:

```
INSERT INTO clima VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ERROR:  insert or update on table "clima" violates foreign key constraint "clima_cidade_fkey"
DETAIL:  Key (cidade)=(Berkeley) is not present in table "cidades".
```

```
-- Tradução da mensagem

ERRO:    inserção ou atualização na tabela "clima" viola a restrição de chave estrangeira "clima_cidade_fkey"
DETALHE:  Chave (cidade)=(Berkeley) não está presente na tabela "cidades".
```

O comportamento das chaves estrangeiras pode receber ajuste fino no aplicativo. Não iremos além deste exemplo simples neste tutorial, mas consulte o Capítulo 5 para obter informações adicionais. Com certeza o uso correto de chaves estrangeiras melhora a qualidade dos aplicativos de banco de dados, portanto incentivamos muito que se aprenda a usá-las.

3.4. Transações

Transação é um conceito fundamental de todo sistema de banco de dados. O ponto essencial da transação é englobar vários passos em uma única operação de tudo ou nada. Os estados intermediários entre os passos não são vistos pelas demais transações simultâneas e, se ocorrer alguma falha que impeça a transação chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de forma alguma.

Por exemplo, considere um banco de dados de uma instituição financeira contendo o saldo da conta corrente de vários clientes, assim como o saldo total dos depósitos de cada agência. Suponha que se deseje transferir $100.00 da conta da Alice para a conta do Bob. Simplificando barbaramente, os comandos SQL para esta operação seriam:

```
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
UPDATE filiais SET saldo = saldo - 100.00
    WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Alice');
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Bob';
UPDATE filiais SET saldo = saldo + 100.00
    WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Bob');
```

Os detalhes destes comandos não são importantes aqui; o importante é o fato de existirem várias atualizações distintas envolvidas para realizar uma operação bem simples. A contabilidade quer ter certeza que todas as atualizações são realizadas, ou que nenhuma delas é realizada. Não é interessante uma falha no sistema fazer com que Bob receba $100.00 que não foi debitado da Alice. Também a Alice não continuará sendo uma cliente satisfeita se o dinheiro for debitado da conta dela e não for creditado na de Bob. É necessário garantir que, caso aconteça algo errado no meio da operação, nenhum dos passos executados até este ponto produza efeito. Agrupar as atualizações em uma *transação* dá esta garantia. Uma transação é dita como sendo *atômica*: do ponto de vista das outras transações, ou a transação acontece completamente ou nada acontece.

Desejamos, também, ter a garantia de estando a transação completa e aceita pelo sistema de banco de dados, que esta fique permanentemente gravada, e não seja perdida mesmo no caso de acontecer uma pane logo em seguida. Por exemplo, se estiver sendo registrado saque em dinheiro pelo Bob não se deseja, de forma alguma, que o débito em sua conta corrente

desapareça por causa de uma pane ocorrida logo depois dele sair da agência. Um banco de dados transacional garante que todas as atualizações realizadas por uma transação ficam registradas em meio de armazenamento permanente (ou seja, em disco), antes da transação ser considerada completa.

Outra propriedade importante dos bancos de dados transacionais está muito ligada à noção de atualizações atômicas: quando várias transações estão executando simultaneamente, cada uma delas não deve enxergar as alterações incompletas efetuadas pelas outras. Por exemplo, se uma transação está ocupada totalizando o saldo de todas as agências, não pode ser visto o débito efetuado na agência da Alice mas ainda não creditado na agência do Bob, nem o contrário. Portanto, as transações devem ser tudo ou nada não apenas em termos do efeito permanente no banco de dados, mas também em termos de visibilidade durante o processamento. As atualizações feitas por uma transação em andamento não podem ser vistas pelas outras transações enquanto não terminar, quando todas as atualizações se tornam visíveis ao mesmo tempo.

No PostgreSQL a transação é definida envolvendo os comandos SQL da transação pelos comandos BEGIN e COMMIT. Sendo assim, a nossa transação bancária ficaria:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
-- etc etc
COMMIT;
```

Se no meio da transação for decidido que esta não deve ser efetivada (talvez porque tenha sido visto que o saldo da Alice ficou negativo), pode ser executado o comando ROLLBACK em vez do COMMIT para fazer com que todas as atualizações sejam canceladas.

O PostgreSQL, na verdade, trata todo comando SQL como sendo executado dentro de uma transação. Se não for utilizado o comando `BEGIN`, então cada comando possui um `BEGIN` e, se der tudo certo, um `COMMIT` individual envolvendo-o. Um grupo de comandos envolvidos por um `BEGIN` e um `COMMIT` é algumas vezes chamado de *bloco de transação*.

> **Nota:** Algumas bibliotecas cliente emitem um comando `BEGIN` e um comando `COMMIT` automaticamente, fazendo com que seja obtido o efeito de um bloco de transação sem ser perguntado. Verifique a documentação da interface utilizada.

É possível controlar os comandos na transação de uma forma mais granular utilizando os *pontos de salvamento* (`savepoints`). Os pontos de salvamento permitem cancelar partes da transação seletivamente, e efetivar as demais partes. Após definir o ponto de salvamento, através da instrução `SAVEPOINT`, é possível cancelar a transação até o ponto de salvamento, se for necessário, usando `ROLLBACK TO`. Todas as alterações no banco de dados efetuadas entre a definição do ponto de salvamento e o cancelamento são desprezadas, mas as alterações efetuadas antes do ponto de salvamento são mantidas.

Após cancelar até o ponto de salvamento este ponto de salvamento continua definido e, portanto, é possível cancelar várias vezes. Ao contrário, havendo certeza que não vai ser mais necessário cancelar até o ponto de salvamento, o ponto de salvamento pode ser liberado, para que o sistema possa liberar alguns recursos. Deve-se ter em mente que liberar ou cancelar até um ponto de salvamento libera, automaticamente, todos os ponto de salvamento definidos após o mesmo.

Tudo isto acontece dentro do bloco de transação e, portanto, nada disso é visto pelas outras sessões do banco de dados. Quando o bloco de transação é efetivado, as ações efetivadas se tornam visíveis como uma unidade para as outras sessões, enquanto as ações canceladas nunca se tornam visíveis.

Recordando o banco de dados da instituição financeira, suponha que devesse ser debitado $100.00 da conta da Alice e creditado na conta do Bob, mas que foi descoberto em seguida que era para ser creditado na conta do Wally. Isso poderia ser feito utilizando pontos de salvamento como mostrado abaixo:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
SAVEPOINT meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Bob';
-- uai ... o certo é na conta do Wally
ROLLBACK TO meu_ponto_de_salvamento;
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Wally';
COMMIT;
```

Obviamente este exemplo está simplificado ao extremo, mas é possível efetuar um grau elevado de controle sobre a transação através do uso de pontos de salvamento. Além disso, a instrução `ROLLBACK TO` é a única forma de obter novamente o controle sobre um bloco de transação colocado no estado interrompido devido a um erro, fora cancelar completamente e começar tudo de novo.

3.5. Herança

Herança é um conceito de banco de dados orientado a objeto, que abre novas possibilidades interessantes ao projeto de banco de dados.

Vamos criar duas tabelas: a tabela `cidades` e a tabela `capitais`. Como é natural, as capitais também são cidades e, portanto, deve existir alguma maneira para mostrar implicitamente as capitais quando todas as cidades são mostradas. Se formos bastante perspicazes, poderemos criar um esquema como este:

```
CREATE TABLE capitais (
  nome       text,
  populacao  real,
  altitude   int,    -- (em pés)
  estado     char(2)
);

CREATE TABLE interior (
  nome       text,
  populacao  real,
  altitude   int     -- (em pés)
);

CREATE VIEW cidades AS
  SELECT nome, populacao, altitude FROM capitais
    UNION
  SELECT nome, populacao, altitude FROM interior;
```

Este esquema funciona bem para as consultas, mas não é bom quando é necessário atualizar várias linhas, entre outras coisas.

Esta é uma solução melhor:

```
CREATE TABLE cidades (
  nome       text,
  populacao  real,
  altitude   int     -- (em pés)
);

CREATE TABLE capitais (
  estado      char(2)
) INHERITS (cidades);
```

Neste caso, as linhas da tabela `capitais` *herdam* todas as colunas (`nome`, `populacao` e `altitude`) da sua tabela *ancestral* `cidades`. O tipo da coluna `nome` é `text`, um tipo nativo do PostgreSQL para cadeias de caracteres de tamanho

variável. As capitais dos estados possuem uma coluna a mais chamada estado, que armazena a sigla do estado. No PostgreSQL uma tabela pode herdar de nenhuma, uma, ou de várias tabelas.

Por exemplo, a consulta abaixo retorna os nomes de todas as cidades, incluindo as capitais dos estados, localizadas a uma altitude superior a 500 pés:

```
SELECT nome, altitude
  FROM cidades
  WHERE altitude > 500;
```

```
   nome    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 linhas)
```

Por outro lado, a consulta abaixo traz todas as cidades que não são capitais de estado e estão situadas a uma altitude superior a 500 pés:

```
SELECT nome, altitude
    FROM ONLY cidades
    WHERE altitude > 500;
```

```
   nome    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 linhas)
```

Nesta consulta a palavra chave `ONLY` antes de `cidades` indica que a consulta deve ser efetuada apenas na tabela `cidades`, sem incluir as tabelas abaixo de `cidades` na hierarquia de herança. Muitos comandos mostrados até agora — `SELECT`, `UPDATE` e `DELETE` — permitem usar a notação `ONLY`.

> **Nota:** Embora a hierarquia seja útil com freqüência, como não está integrada às restrições de unicidade e de chave estrangeira, sua utilidade é limitada. Consulte a Seção 5.5 para obter mais detalhes.

3.6. Conclusão

O PostgreSQL possui muitas funcionalidades não abordadas neste tutorial introdutório, o qual está orientado para os usuários com pouca experiência na linguagem SQL. Estas funcionalidades são mostradas com mais detalhes no restante deste livro.

Se for necessário mais material introdutório, por favor visite o sítio do PostgreSQL na Web (http://www.postgresql.org) para obter indicações sobre onde encontrar este material.